TAMPERE UNIVERSITY OF TECHNOLOGY

JOONAS KYLÄNPÄÄ

AUTOMATED PERFORMANCE MEASUREMENTS IN MOBILE LINUX DEVICE

Master of Science Thesis

Examiner: Professor Hannu-Matti Järvinen
Examiner and topic approved in the Faculty of Computing and Electrical Engineering Council meeting on 5 May 2011

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Engineering
KYLÄNPÄÄ, JOONAS: Automated performance measurements in mobile Linux device
Master of Science Thesis, 44 pages, 1 Appendix page
June 2012
Major: Software engineering
Examiner: Professor Hannu-Matti Järvinen
Keywords: Testing, test automation, mobile device, performance, current measurement

As mobile devices get more features and operating systems get more complex, testing becomes more important. When there are more and more complex features, it means that testing needs to be even more thorough. This means more test cases, more time spent testing and more money spent on testing. Costs for testing can be reduced either by reducing amount of test cases or adding automation to testing process.

Benefits for automating testing are clear. More tests can be longer and run more often. Automation also removes human errors from test results. There is no need to use test engineers to run the tests. Test cases are always run exactly the same way and only need to be evaluated afterwards. Automation needs purposely made environment to work. There are more overhead to start testing, but once environment is set up and changes needed to run fully automated test cases are taken into account, benefits of test automation can be seen.

This thesis introduces MeeGo QA tools tool chain, automates currently partly manual performance test asset to be fully automated and adds support for measurements to Testrunner-lite. Testrunner-lite is part of MeeGo QA tools tool chain and handles execution of test cases according to test plan XML file. Measurement support for other tools was done by different people and is not part of this thesis. At the start, test cases were already automatic, but needed manual effort to setup the testing.

At the end of this thesis project, every part of performance testing is fully automated. Test run triggers at certain schedule, build system prepares image, OTS runs the tests using Testrunner-lite and reports results through QA Reports. Measurement results are reported as part of the test results. Tests can still be run, although the number of test engineers is lower.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelun ja tietoliikennetekniikan koulutusohjelma
KYLÄNPÄÄ, JOONAS: Automaattiset suorituskykymittaukset Linux mobiililaitteessa
Diplomityö, 44 sivua, 1 liitesivu
Kesäkuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen
Avainsanat: Testaus, testiautomaatio, mobiililaite, suorituskyky, virtamittaus

Kun, mobiililaitteiden omnaisuuksien määrä kasvaa ja käyttöjärjestelmät tulevat monimutkaisemmiksi, testaamisesta tulee yhä tärkeämpää. Kun järjestelmässä on enemmän monimutkaisia ominaisuuksia se tarkoittaa, että testauksen tulee olla kattavampaa kuin ennen. Tämä tarkoittaa enemmän testitapauksia, jolloin testaukseen kuluu enemmän aikaa ja rahaa. Testaukseen liittyviä kuluja voidaan alentaa vähentämällä testitapauksia tai käyttämällä testiautomaatiota.

Automaattisen testauksen tuomat hyödyt ovat selviä. Testit voivat olla pidempiä ja niitä voidaan ajaa useammin. Automaatio myös poistaa inhimilliset virheet testausprosessista. Testit ajetaan aina samalla tavalla ja vaativat vain tulosten analysoinnin jälkeenpäin. Testiautomaatio tarvitsee erikseen rakennetun ympäristön. Automaattisen testauksen aloittamiseen liittyy ylimääräistä työtä, mutta kun ympäristö on pystytetty ja muutokset, joita täysin automaattiseen testaukseen tarvitaan, on otettu huomioon, voidaan nähdä automaattisen testauksen hyödyt.

Tässä diplomityössä esitellään MeeGo QA tools työkaluketju, automatisoidaan osittain manuaalinen suorituskyky testipaketti ja lisätään Testrunner-lite komponenttiin tuki mittaustulosten käsittelyyn. Testrunner-lite on osa MeeGo QA tools työkaluketjua ja sen tehtävä on hoitaa testien suoritus testisuunnitelma XML-tiedoston mukaisesti. Mittaustulosten käsittely työkaluketjun muihin työkaluihin ei ole osa tätä diplomityötä. Aloitettaessa diplomityötä testit olivat jo automaattisia, mutta testiympäristö piti alustaa manuaalisesti.

Tämän projektin lopussa jokainen suorituskykytestien osa-alue on täysin automatisoitu. Testiajot ajetaan tietyn aikataulun mukaan automaattisesti, käytettävä ohjelmisto luodaan automaattisesti, OTS ajaa testit automaattisesti käyttäen Testrunner-lite työkalua ja tulokset raportoidaan automaattisesti QA Reports -palvelun kautta. Testien suorittaminen ei kärsi, vaikka testaajien määrä on vähentynyt.

PREFACE

This thesis work was carried out for Nokia Corporation during 2011 and 2012. The examiner for this thesis was Professor Hannu-Matti Järvinen.

I would like to thank MeeGo QA Tools team for being a great team to work in and for all the help I got from the team members during this process. Also thanks to Core System Testing team for making the test cases and Nokia Corporation for providing the facilities.

Finally, I would like to thank Kirsi Tirronen for supporting me during the final crunch of writing this thesis.

Tampere, 7 May 2012

Joonas Kylänpää

TABLE OF CONTENTS

ABBREVIATIONS

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CI	Continuous integration
CSV	Comma-Separated Values
DHCP	Dynamic Host Configuration Protocol
FTP	File Transfer Protocol
HAT	Hardware Accessory for Testing
HTTP	HyperText Transfer Protocol
MWTS	MiddleWare Test Suite
NFT	Non-Functional Testing
OTS	Open Test Service
QA	Quality Assurance
RTSP	Real Time Streaming Protocol
SIP	Session Initiation Protocol
SSH	Secure Shell
TDD	Test driven development
UI	User interface
VCS	Version Control System
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language Remote Procedure Call
WLAN	Wireless Local Area Network

1. INTRODUCTION

Mobile devices are getting closer to fulfilling the same tasks as a desktop computer. Processing power increases and this makes it possible to make more features and more complex functionality. When new features are added, the whole software on the device is getting bigger and more complex. Software testing aims to go through all the features and test that they work correctly.

Larger software means that time needed to testing increases. To stop the costs from rising too much, two things can be done: either reduce the amount of test cases or use test automation. Obviously decreasing the test case count does not sound like a good idea. Decreasing the number of test cases lowers the test coverage and makes it harder to be sure that the system works correctly. Although this may decrease the testing cost, selecting test cases to be left out can be difficult while still providing enough test coverage. On the other hand, test automation makes it possible to add even more test cases while saving test engineers time. Once the test automation system is up and running, it is possible to add more test cases without increasing the cost of testing.

Earlier, when development of system was still on full swing testing was done manually by large number of test engineers. Now that the project is coming to a close there are less test engineers left. This automation effort was made to support the decreasing number of test engineers.

The device discussed in this thesis is running operating system based on Linux. Also all of the tools described are designed to run on Linux. Performance is important feature in today's mobile devices. Performance plays big part in the user experience conveyed to the user. The process of automating performance tests was started from existing partly automated test cases. The goal was to make running test cases fully automated without any need for human interaction. These tests will be run weekly to observe the quality of weekly releases.

When starting this thesis, test automation tools did not support displaying of measurement results. Support for gathering measurements from the test assets was added. This was done by specifying a new feature to the test plan to be used by test assets. After this, support for measurements was implemented across the whole tool chain. This thesis focuses on measurement support implementation of Testrunner-lite. Testrunner-lite is part of MeeGo QA tools tool chain and handles execution of test cases according to test plan.

The thesis is organized as follows. Theory of software testing is introduced in Chapter 2, and it provides information about different testing levels and practices related to software testing.

Chapter 3 introduces the MeeGo QA tools that are the tools used in test automation. This includes tools used to plan test cases, run them and finally reporting the results. Also the hardware environment needed is described in this chapter.

Chapter 4 is the place where the actual work in this thesis has been done. It explains the format of the test assets implemented. Implementing measurement support to test plan and Testunner-lite is explained in great detail. Finally we take a look how the tests are run, how results are reported, and analyze the results.

Chapters 5 contains some conclusions about the whole process.

2. AUTOMATED TESTING

Software testing is often under-valuated and not well-liked part of software development process. Developers generally do not like writing and running tests as it can feel repetitive and boring. Programs and operating systems are getting bigger and more complex all the time as more features get added. This means that testing is becoming more important. Testing needs to paid more attention than previously. As this leads to increased costs and more time being consumed, the testing process needs to be developed further. This has advanced the development of test automation.

Different phases of software testing can be divided to four parts [1]:

- Planning the testing.
- Creating the environment for testing.
- Executing the test cases.
- Evaluating the results.

This thesis goes through these phases from performance testing point of view. Another focus is on test automation tools used to run the tests.

The goal of software testing is to find defects from the features that the software implements. Of course it is not possible to test everything when software gets bigger than few lines long. There should be special attention paid to the testing process and focus on delivering as good coverage as possible with the test cases. Available resources determine how much testing can be done. Test automation aims to help make testing easier and less time consuming and freeing test engineers to other task than the test execution itself.

2.1. Software testing

Testing can be divided to different levels. For example in commonly known V-model testing is divided in 3 parts which are shown in Figure 2.1. Planning the testing is started at the same time as planning of the software. System testing is designed when requirement specification of the program is being made. Integration testing is done on architecture design phase. Finally module testing is specified when module design is being done.

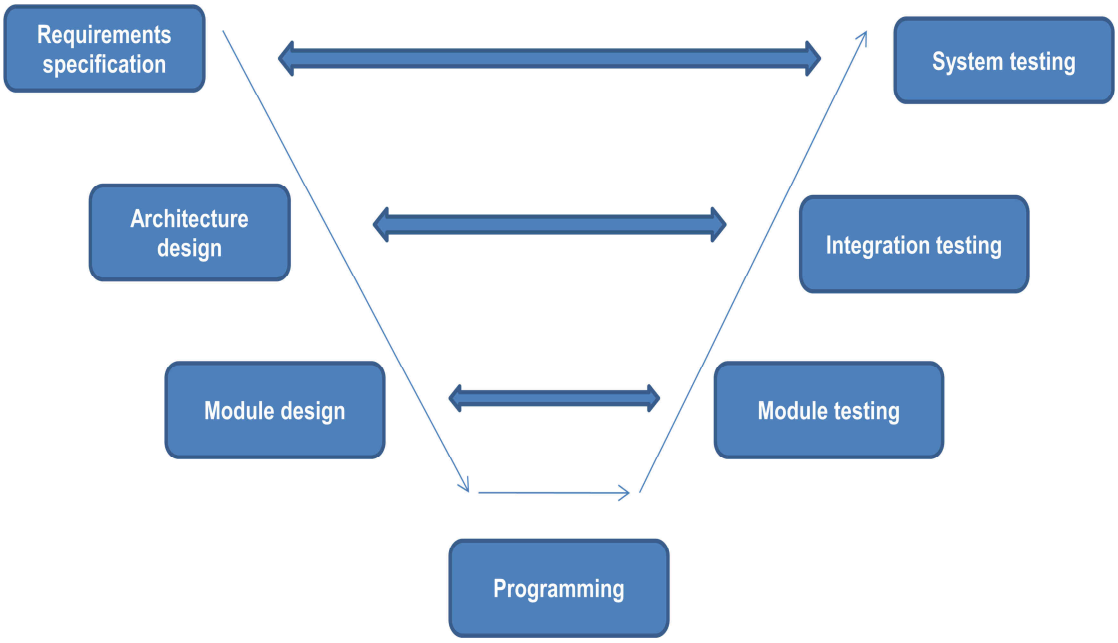


Figure 2.1 V-model in testing [1]

V-model separates three different levels for testing:

- System testing.
- Integration testing.
- Module/Unit testing.

The testing side of V-model starts from the module testing, which is also called unit testing. In this phase, particular modules of the program are tested. This phase goes through the functionality of the module and makes sure that all the parts of the module are working correctly and handle error situations properly. Usually unit tests are written by the developer of the module itself. This way the person doing the testing has the best knowledge of the module and knows what and how it needs to be tested.

After module testing there is integration testing level. This tests how components interact with each other. Integration testing is introduced in more detail in the next chapter.

The last level is called system testing. In this phase, whole system, meaning all the components together are verified to work correctly. While module testing and integration testing are usually done by the team developing modules, system testing is done independently of the development by dedicated team of test engineers. System testing is introduced in Chapter 2.1.2.

2.1.1. Integration testing

In integration testing, the target is to make sure that the component works correctly with other components in the system. This phase is performed before new version of the

component is integrated and taken into use in the system. As module testing is done before this phase, inner working of the new component should already be verified before this phase. Architecture design shows the components that modified component interacts with. This can be used to plan the testing and analyzing the results.

Goal is to have only one or few modified components under testing at once. This way testing can focus on those and their interaction with other modules. System testing is done on a higher level and cannot verify all the paths through the software. This is the job of integration testing.

2.1.2. System testing

System testing is usually done by dedicated team that is not directly involved in development of the software. The whole system with all its features are under test as thoroughly as possible. System testing is needed because individual testing of components cannot guarantee that the whole system works. System testing phase failures can result from interactions that cannot be produced without all the components being exercised in the target environment [2]. Although the modules being tested have already passed integration testing, system testing is still needed for the exact reason stated above.

Results from system testing are compared to the requirement specifications and all the requirements should be fulfilled. System testing is usually black-box testing done by teams not familiar with the actual code. Black-box testing is introduced in chapter 2.1.4. Some of the system testing subtypes [3]:

- (non-)functional testing
- regression testing
- security testing
- stress testing
- performance testing
- usability testing
- random testing
- data integrity testing
- conversion testing
- backup and recoverability testing
- configuration testing
- operational readiness testing
- user acceptance testing
- alpha/beta testing.

This list shows the wide variety of testing that system testing can cover. Not all the types are covered in this thesis, but are shown only as reference.

2.1.3. Regression testing

Regression testing is an important phase in the test process. When a new version of a component is integrated it should be made sure that all the old features are still functioning as before. Unit testing aims to verify that module works by itself but also there is need to verify the functionality with other components. A modified component can fail when used with unchanged components. Failures can be caused by some kind of unwanted side effects or changes in communication between components. When this kind of behavior happens, it is considered as regression and regression testing is trying to solve this problem [2].

When new functionality or features are added to a component, there should be a new test case to test the feature. Also other already implemented features should be tested after new functionality is introduced. After the new feature and all the old features seem to be ok, this new functionality can be released and new test case added to the regression test set. This set should contain tests for all the features implemented to the software.

When more features get added the regression test set can grow quickly. At some point running all the regression tests can become very time consuming. This is why the regression testing is the area where automation offers greatest return on investment [3]. When regression test cases are automated, they can be run anytime without limiting test engineers other tasks. This way set can be made bigger and run more often as new features get implemented. When tests are automated, all the tests are run every time and not skipped because of lack of time or other excuses.

As new features get added, the regression test set needs to be constantly maintained. In addition to adding new test cases, old ones should be evaluated from time to time to remove overlapping test cases. When number of test cases becomes very large, time needed to run the test cases may become problem even with test automation. Binder provides an example where 90% of 165000 test cases were redundant. Added to this, the test coverage was also lacking. After evaluation and rewriting some test cases total count of test cases was around 18000 and coverage had increased [2].

Regression testing should happen on all test levels. Testrunner-lite’s regression tests are on the unit test and system test level. Testrunner-lite and other MeeGo QA tools are not part of the mobile device’s operating system. The tools are part of the support infrastructure and QA tools team handles system testing of all the MeeGo QA tools. This is different to how system testing is normally done by a dedicated testing team. For all new Testrunner-lite features and functionality unit and regression tests are added.

2.1.4. Black-box testing vs. white-box testing

Software testing can be divided to black-box and white-box testing. Black-box testing is done without knowing how the module or application works internally. White-box testing means that the person testing is familiar with the code and its behavior.

White-box testing is usually done by developers of the module. It may be that module has no public interfaces and only communicates to some other module. When developer is familiar with the inner workings of the module, he is able to verify all those private connections. White-box can also be automated with suitable tools that, for example, seek memory leaks and other possible failures.

Black-box testing is usually done by someone not knowing the inner workings of the module. The goal of black-box testing is to test that module produces correct results for given inputs and doing so verify that public interfaces are working.

These methods can be used in different testing levels. Module testing is usually done as white-box testing. This is because usually the developer is responsible for the module testing phase. System testing level is usually done as black-box testing by dedicated testing team without knowledge of inner working of the modules under test. Integration testing can be harder to put into these two categories. Integration testing can be considered grey-box testing that is between white- and black-box testing. As integration testing is usually done by test engineers working closely with the team that develops the modules, he or she has more knowledge about the module under test than the team doing system testing.

2.1.5. Performance testing

Performance testing aims to determine the responsiveness, throughput, reliability or scalability of the system. Some of the goals of performance testing are [6]:

- Assess production readiness.
- Evaluate against performance criteria.
- Compare performance characteristics of multiple systems or system configuration.
- Find the source of performance problems.
- Support system tuning.
- Find throughput levels.

The goal of the performance testing is different from normal testing that aims to find defects. Performance testing aims to identify performance bottlenecks and establish a baseline for future regression tests. Software under test should be stable enough so that testing process runs smoothly on the system.

Software performance is a critical point in the success of the software. Performance plays a big role in the user experience. Performance testing can help the programmer to assess the readiness of the system by providing data indicating the likelihood of user being dissatisfied to the performance characteristics of the system. This data can then be evaluated against performance criteria.

Another goal is to find the sources of performance problems by identifying bottlenecks in the software. Performance optimization should only be done after performance problems are identified. Premature optimization can lead to unclean and messy code which in turn makes maintaining and extending the code harder. Only after the performance bottlenecks are identified, optimization can be made.

This thesis focuses on benchmarking part of the performance testing. Benchmarking compares system's performance against a baseline that has been created internally, or against an industry standard. Measuring WLAN (Wireless Local Area Network) throughput level or SMS (Short Message Service) message sending latency can be considered benchmarking.

2.2. Automating the testing process

Automation aims to save time and money used in manual testing. Less time spent on testing means more time to improve the test coverage and investigate other problems. There can be large overhead when starting the automating process when test engineers are unfamiliar with the tools or tools are not stable and reliable enough. Over time the savings made by automation will be larger than the extra effort used to start test automation. Also better testing will lead to better software, because defects are discovered more reliably and earlier.

2.2.1. Benefits of test automation

Test automation needs extra consideration and planning. If taken into account, the test automation can provide several benefits. These can be achieved, for example, with easily reproducible tests and reduced manpower to run test cases. Three significant benefits for test automation have been identified [3]:

1. production of reliable system,
2. improvement of the quality of the test effort, and
3. reduction of the test effort and minimization of the schedule.

Producing a reliable system is one of the most important features for end user. Application crashing or slow user interface create poor experience for the user. User should also be confidently able to trust that the data in the device is safe and that it will not be lost due to some unexpected malfunction.

Performance and reliability testing are used to verify the performance and stability of the system. Performance testing goes through features and checks the performance of a particular task. It aims to show the system's ability to perform tasks in certain time. For example, when testing multimedia features, suitable targets could be testing playback of various video formats and recording the playback FPS (Frames Per

Second). This could be done manually but that would be subjective and testing large number of different formats would take a long time.

With reliability testing, one test case is run multiple times iteratively. This means, for example, opening a WLAN connection, downloading same file over WLAN connection 100 times and disconnecting from the wireless LAN connection. Connecting to a WLAN and downloading should work every time. Doing such a repetitive task so many times would take really long time to do manually. It would also be really boring and redundant for the test engineer doing it. Over such many repetitions, mistakes would be likely to happen.

Both of these problems can be solved by using test automation. The test runs are always run exactly the same way and can be compared to previous results.

The second benefit mentioned is improvement of the quality of the test effort. This can be difficult to do without increasing the cost of testing. Test automation aims to make this easier. Test effort quality can be improved by build acceptance and regression tests. Testing done during build acceptance focuses on major system functionalities when a new software build is received. When using continuous integration new builds can be built fairly often. Instead of running same tests manually for every build, they can be automated and run more efficiently and often. After testing that the basic functionality on the build is working more thorough testing can be done.

Regression testing can be run after build acceptance test has passed. Regression testing in system testing level is run after new image is released. All the old features are tested to still work when introducing new code. Regression test expands build acceptance testing and instead of testing just basic functionalities, tests all the functionality that has already proved to work on the previous builds. This is typically time consuming and great benefits can come from automating regression testing.

Automation can also make running tests easier on multiple devices or software platforms. If there are multiple devices being developed at the same time, tests need to be run on all of them separately. In addition to that, there could be different configurations of the image, like different variants for different countries. These different devices and configurations multiply the time needed for manual testing.

Especially between different country variants there is a good chance that the same testing scripts can be reused for all the different configurations or at least modified to be reusable with input parameter changes. Test cases for other configuration can be run on the same time as analyzing the results from previous runs. Best case scenario would be if the test automation environment provides the functionality to run multiple variant tests at the same time.

Automation also eliminates the variance in test results caused by different people running the same tests. Automatic environment always runs the tests exactly the same way and is always able to produce consistent results. Test automation system can also provide possibility to schedule running tests. For example, this way tests can be triggered at night and results will be ready by the morning when test engineers come to work. This is especially helpful if tests take several hours.

As mentioned in the previous chapter, the test automation can reduce the workload of test engineers and still increase the coverage of testing. However, when planning test automation the workload needed to start the testing has to be taken into account. Study about the test automation showed that the test automation consumed only 25% of the man-hours needed for manual testing [3]. How much of the tests is possible to automate varies between different systems, so this number is the best case when there is a possibility to automate most of the tests. This shows a huge potential of time saving with test automation.

As expected, biggest savings were in test execution time. Not only are the automated tests faster to run, but test engineer is also free to do other tasks during test execution. Other big saving was found in analyzing and reporting the results. Test automation tools can produce reports automatically and send them to reporting tools automatically. These tools usually provide history data which makes comparing difference between test run results easier.

2.2.2. Drawbacks of test automation

Test automation will not automatically solve all the testing related problems. Some of the functionality may still need to be tested manually. When developing consumer electronics device, user experience of the device is very important when making the decision to buy the new device. While features can easily be listed and they can be tested automatically, the actual usability of the device can be very hard to test automatically.

There are frameworks that make it possible to test the user interface and even measure the time between user interface transitions. UI (User Interface) specifications that are available can be used when developing automated UI test cases. However, there are some cases that cannot be tested automatically. For example, there may be tearing on the screen during UI transitions. Even when the measured time between UI transitions is fine, it may look bad to the user. This is example of something that is impossible or very difficult to detect automatically, and manual testing is needed.

Another difficult thing to test is the logic how the user interface works. When automatic test cases are made based on different documentation, it is difficult to piece together how the system actually works. These sorts of usability issues may be immediately visible when using the device manually.

Great reproducibility is a benefit for test automation, but it can also be seen as a drawback. The tests are run exactly the same way and in exactly the same order without any variance between test runs or test steps. When doing manual testing there can be little variation depending on the person doing the testing. Test engineer can, for example, be distracted during testing and press the wrong button. This might reveal some issue that might not have otherwise been seen.

Taking automated testing into use in an organization should be considered carefully before starting the project. There may be false expectations in management how the

automatic system will work and what it does to the testing process. Some of these false expectations are listed here [3]:

- automatic test plan generation
- test tool fits all
- immediate test effort reduction
- immediate schedule reduction
- tool ease of use
- universal application of test automation
- one hundred percent test coverage.

Some of the expectations listed can seem little bit stupid, but for person not being too much involved in testing or test automation tools, it might seem plausible. For example, there will always be need for multiple tools as otherwise the tool would become unnecessarily complicated. For example in this thesis, the performance testing is done using middleware test suite and UI testing is done by different tools and test suites.

The test effort and schedule reduction are not likely to happen immediately. When new tools are taken into use, there are multiple things that are likely to slow down the testing before making it faster. First, the test automation environment with all the tools in use needs to be built. When this is done, it is likely that the tools are not stable and reliable enough, also some of the needed features may be lacking. Second thing that needs to be taken into account is that even though tools and environments used would be stable and reliable, test engineers need to learn how to use the test tools. Test cases may need to be developed with completely new tools. As starting the automation can take time, manual testing needs to be still done while learning to use test automation tools. This can actually delay the schedule and make the situation worse than before automation effort was started.

After the initial problems are solved, the automatic test set can keep growing and benefits will start to become visible. However, no matter how much the automated test set grows it is still impossible to achieve one hundred percent test coverage. If we think about wireless LAN performance testing we should test this against every possible access point and every possible data going through that connection. It is easy to see that we cannot test our device with every access point in the world and it is also impossible test every possible data variations going through the connection.

2.2.3. Continuous Integration

If components are integrated to the system rarely, this can introduce very big changes and the possibility of something not working becomes more likely. This is the traditional approach to integration; integration process has been done once a week, once a month, or even less frequently. When systems become larger, this will introduce a lot

of problems. This can lead to problems fitting different modules together or even compiling a new version of the software. Also developing test cases can be different. Test engineers need to take into account all the new features and modified modules and try to be able to test all that for one single release. With multiple changes in code it is very hard to predict how long the integration process will take. In the late phases of software development this can become a major problem and effect deadlines. It is also hard to see the current state of the program, when no new features have been integrated in the system for a long time [4].

Continuous integration aims to help both integrators and test engineers. The idea is to integrate new components to the system as often as possible, even several times a day. When developer commits new code to the version control system, an automated build process should start immediately and provide fast feedback to the developer. As this process is the same for all teams, each team now has a common understanding what constitutes of successful build. When tests are run immediately on a few commits, possible defects get noticed faster and are easier to fix. System is also up to date all the time as new components are constantly integrated. This also makes it easier for everybody involved to see the current status of the system and the features integrated.

New software images are done very frequently and this has a big effect on the testing process. For each new version of a component there needs to be integration tests that get run before component is accepted. If the build process of new image is automated, also the testing should be automated and included in the same process. Image including the new component should build automatically and after building, tests should be executed on the image. This makes it possible to test that the component works before the actual integration of a component

In case of this thesis, when releasing a new version of either Testunner-lite or test asset packages no manual intervention is needed. Build system automatically notices new tagged commits and builds a new version of image and runs integration tests.

3. TESTING ENVIRONMENT FOR AUTOMATED TESTING

Testing environment explained in this chapter is originally developed to be part of a continuous integration system. Build system is designed to do build automation that helps developers in their day-to-day activities. In this context build system is responsible for building the software and creating the software images. Build system uses VCS (Version Control System) hooks to monitor new tags. After new tag is recognized, the build system builds a new image with the new component. Repositories store already integrated components and all the other parts of the system are pulled from there. After the image creation succeeds, the new image including the automatic test cases is sent to OTS (Open Test Service) server. Because the build system used was a proprietary system it is not explained in greater detail. Parts involving build system like image creation, creating requests, and exact request format is omitted.

OTS runs the provided tests and reports the results to the user. The results are stored in QA Reports reporting tool, and the user is notified whether the tests were successful or not. If everything seems ok and the tests pass, the new components are stored to the repositories and made available to other teams when they create their own images for testing.

3.1. MeeGo quality assurance tools

MeeGo quality assurance tools are developed to ensure MeeGo software quality. Two main targets are improving test automation and test reporting. Test automation for MeeGo release engineering, and test reporting for MeeGo quality assurance. The goal is to be able to test staging packages often and to automatically ensure the quality of weekly or nightly release. In this chapter, all the components, hardware and software, needed for fully automated testing are introduced. Figure 3.1 shows the software components of MeeGo test automation, starting from the tools used in planning phase and ending to the reporting phase tools in the bottom.

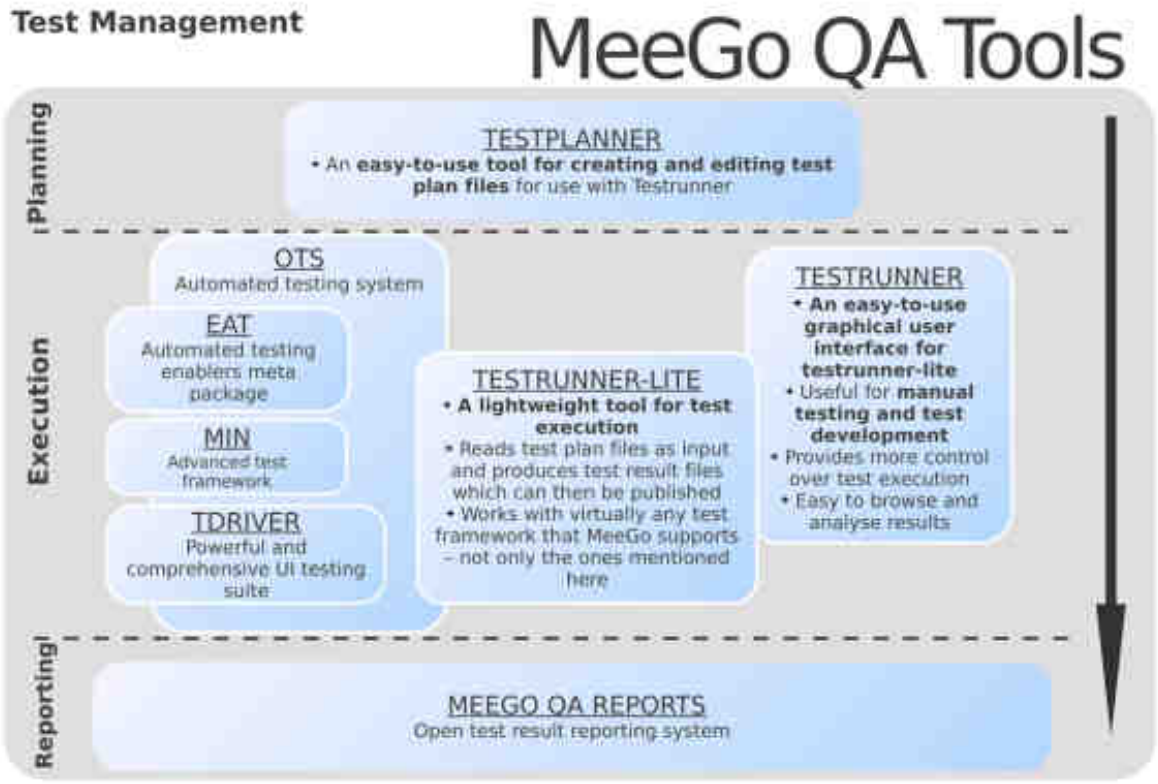


Figure 3.1 Overview of MeeGo QA tools

3.2. Creating and editing test plans

In this context, a test plan means test XML (Extensible Markup Language) file that can be run using Testrunner-lite. This XML file is central to the test automation process. It defines the tests to be run. The basic principle of test plan XML file is that any executable can be used for testing. In case of automated testing, test results are checked from exit codes and in manual testing prompted from user.

3.2.1. Test plan format

The test plan XML is started with the mandatory XML and test-definition elements as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
 <testdefinition version="1.0">
 </testdefinition>
```

Within testdefinition there can be multiple suites containing sets which in turn contain cases. Grouping cases to sets and suites makes test plan more organized. Test cases that test the same feature are usually grouped into one set and sets that test the same architectural domain under the same suite. Within case element are step elements which are the actual commands that will be run. Example of such hierarchy:

```
<?xml version="1.0" encoding="UTF-8"?>
<testdefinition version="1.0">
   <suite name="my-multimedia-tests" domain="Multimedia">
       <description>Video playback tests</description>
       <set name="video-playback-tests" feature="Video Playback">
          <description>Video playback tests</description>
          <case name="playback1" type="Functional" level="Component">
               <step>execute_playback_test</step>
          </case>
       </set>
       <set name="video-recording-tests" feature="Video Recording">
          <description>Video playback tests</description>
          <case name=”recording1” type=”Functional” Level=”Component”>
               <step>execute_recording_test</step>
          </case>
       </set>
   </suite>
</testdefinition>
```

Suite, set, and case have various attributes that can be used to show information about the test plan. The only required attribute is name, but to better show what is being tested some of the optional attributes should be filled. Description element is good for documenting what is being tested. Other attributes like domain are for automatic grouping of test results. Domain attribute is usually on suite level. Domain tells which architectural domain the tests are focused. MeeGo architectural domains can be found from MeeGo developer documentation [6].

Other useful attributes for managing tests are feature, component, type, and level. A feature links the test case to the feature IDs from feature/bug tracker. A component attributes like domains are also documented in the MeeGo developer documentation. A type attribute shows the type of testing, e.g. performance, or security. A level attribute can have values like system or feature, where feature means feature acceptance testing and system means system level testing based on use cases.

3.2.2. Testplanner

The testplanner is a graphical tool for creating test plans. It is developed using Qt framework. The design goal was to make creating test plans easier for people not familiar with test plans or XML format in general. It supports all the attributes in test plan, and feature attributes are fetched straight from feature/bug tracker.

Main window consists of two parts: the left side has a tree view for the suite/set/case hierarchy and the right side an editor for values of a given level. Figure 3.2 shows Testplanner with case level editor opened. All the different hierarchy levels have different settings, so editor view changes depending on what level is selected. Editor has undo/redo functionality for better usability. Tree view allows the users to reorganize elements by drag-and-drop. Testplanner does validation of against test plan XML schema when saving the test plan. An XML schema is a description of a type of XML file. Location of schema to validate against can be chosen from the settings menu.

Default schema to use is the stricter version of test plan. Test engineers can also invoke Testrunner from Testplanner and start executing tests. Testrunner, and the execution of test plan is introduced in the next section. Additional features include integration with Git version control system. This allows users to open test plan directly from Git repository of test package. This feature supports basic Git features like pull, push and commit.

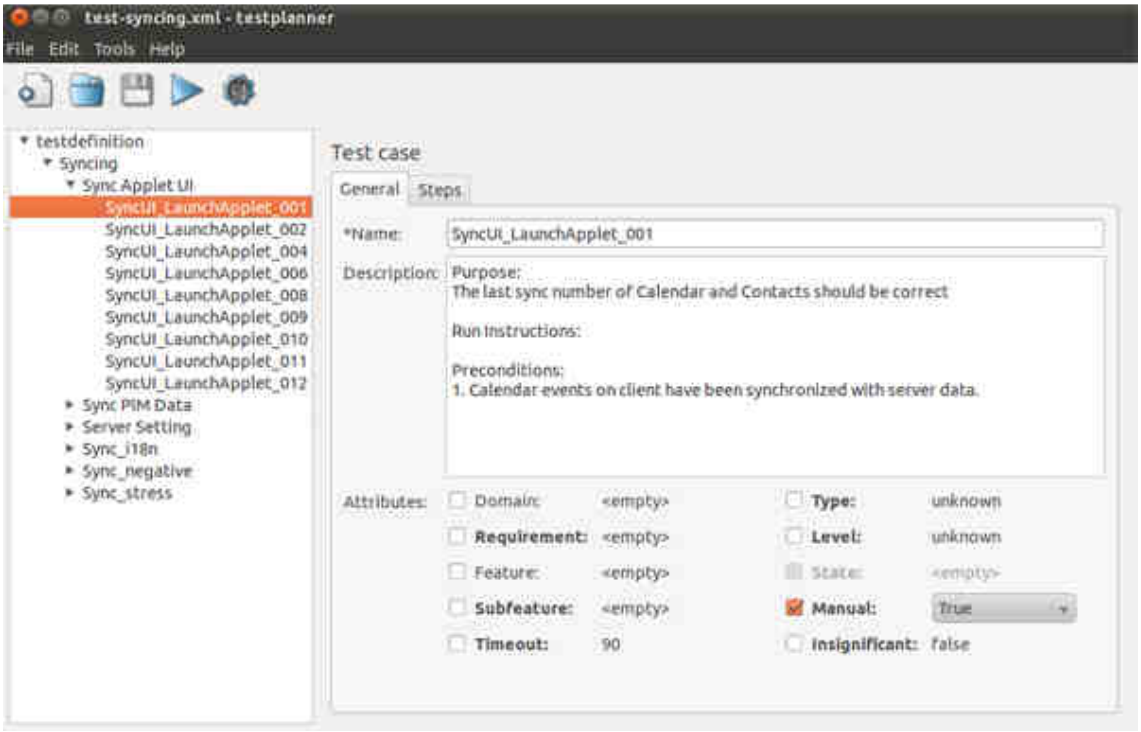


Figure 3.2 Testplanner with case level open

3.3. Testrunner-lite, software to execute the test cases

Testrunner-lite is a light tool for test execution which reads the test plan XML file as input and produces result XML as output. It is a C rewrite of old Python version called Testrunner. Rewrite was done to allow Testrunner to be executed on device, which has considerably less processing power than a desktop PC. Testrunner-lite requires fewer dependencies to be installed than the Python version. Since the rewrite, a lot of new features have been added to Testrunner-lite. This section introduces the main features and functionality of Testrunner-lite.

3.3.1. Test execution and test case verdict

Possible verdicts that Testrunner-lite supports are PASS, FAIL, and N/A. On code level Testrunner-lite treats set elements separately. A set consists of case, pre_steps and post_steps elements. Cases, pre_steps and post_steps consist of one or more step elements. Pre_steps and post_steps are meant for initializing and tear down. They are

executed before and after cases. Failure in pre_steps will cause the whole set to fail and any cases inside it will not be run. Failure info indicating that pre_steps failed will be written for each case. Post_steps will not cause test set failure. One failing step will cause the whole case to fail. After the first step failure in a case, rest of the steps in the case are not run and execution is continued from the next case. FAIL verdict can be caused by either unexpected return value or test step timeout. Test case verdict is N/A if case is marked to be in design state or case has no steps. Testrunner-lite tries to align with agreed MeeGo test case verdicts [7].

3.3.2. Manual tests

The main focus of Testrunner-lite is fully automated testing, but it also supports running manual tests. Manual tests produce the same result XML files and the report can be uploaded to reporting tool manually. When running manual tests, Testrunner-lite outputs case and step description to standard output. Step description in this case is the text inside step element. Instead of runnable command this usually includes longer description what should be done. Then Testrunner-lite prompts for step result which user can input by pressing p, f, or n for PASS, FAIL and N/A respectively. After the whole case is run, there is a possibility to enter additional comments related to the just run test case.

3.3.3. Host-based testing

Testrunner-lite is usually used in a way called host-based testing. This means executing Testrunner-lite on PC and test steps over SSH (Secure Shell) connection on the device. Host-based testing is used in test automation, because if device crashes, the result XML file is still got. This is especially helpful in early days of operating system development when the target device is not yet stable.

There are two types of SSH connection methods available. The first one uses OpenSSH client to connect to device and execute tests. Second method uses libssh2 library. Both methods require key-based authentication between the host and a device. Folder path to private SSH key can be given as parameter to Testrunner-lite. Default is to try to find the key from user's home folder. Libssh2 implementation is faster, when there are a lot of steps in test plan. This native SSH connection method is faster, because there is no need to open new SSH channel between each step. When OpenSSH client is used, it has to open new connection to the device between every step. Libssh2 implementation is still at experimental state, so the slower OpenSSH is the default choice. Executing tests remotely over SSH connection requires extra error handling effort. Testrunner-lite supports resume feature if connection failure happens. It has two parameters for resume, continue and exit. When exit parameter is given Testrunner-lite executes post_steps and get file operations that download possible log files from the device. Continue does the same but instead of exiting after post_steps and get file operations it continues test case execution from the next set.

3.3.4. Host and Chroot testing

Chroot testing (sometimes called host testing) means that test cases are executed on the PC rather than on the device. This allows testing boots, reboots, flashing, controlling extra test equipment like WLAN, Bluetooth devices etc. In addition, it can speed up functional UI tests implemented with the Testability Driver framework by executing the test script on the host and only the software under test (SUT) on the device.

Two variants of this kind of testing are available, native and chrooted. In native mode, test packages are installed to and executed directly on the host. This mode is not usually used, because problems with test packages can negatively affect functionality of the host PC. This can work in controlled situations, but usually it is better idea to use chroot variant.

In chrooted mode, test packages are executed inside a sandbox that can contain any needed software. This sandbox is called rootstrap. With MeeGo image creation tools, any image for i386 architecture can be used to create a rootstrap. This allows host testing to be executed on any host PC and no additional software needs to be installed to PC [8].

3.3.5. Testrunner, graphical interface for testrunner-lite

Testrunner, which confusingly has the same name as the legacy Python-based Testrunner, can be considered as graphical wrapper for Testrunner-lite. It runs the command line client on the background. It is developed using Qt framework. Similarly to Testrunner-lite, it takes test plan XML file as input and produces result XML as output. Interface consists of dock widgets that can be freely moved around. Figure 3.3 shows the opened test plan before any tests are run. Graphical interface is mainly designed to be used with manual testing. It shows more information about test plan and makes entering and editing manual results easier. Test engineer can also choose which tests to execute by selecting tests with checkbox. Like Testplanner, Testrunner can change the order of tests with drag-and-drop. While running, Testrunner shows the progress of test run by updating results to tree view and showing progress bar with percentage value.

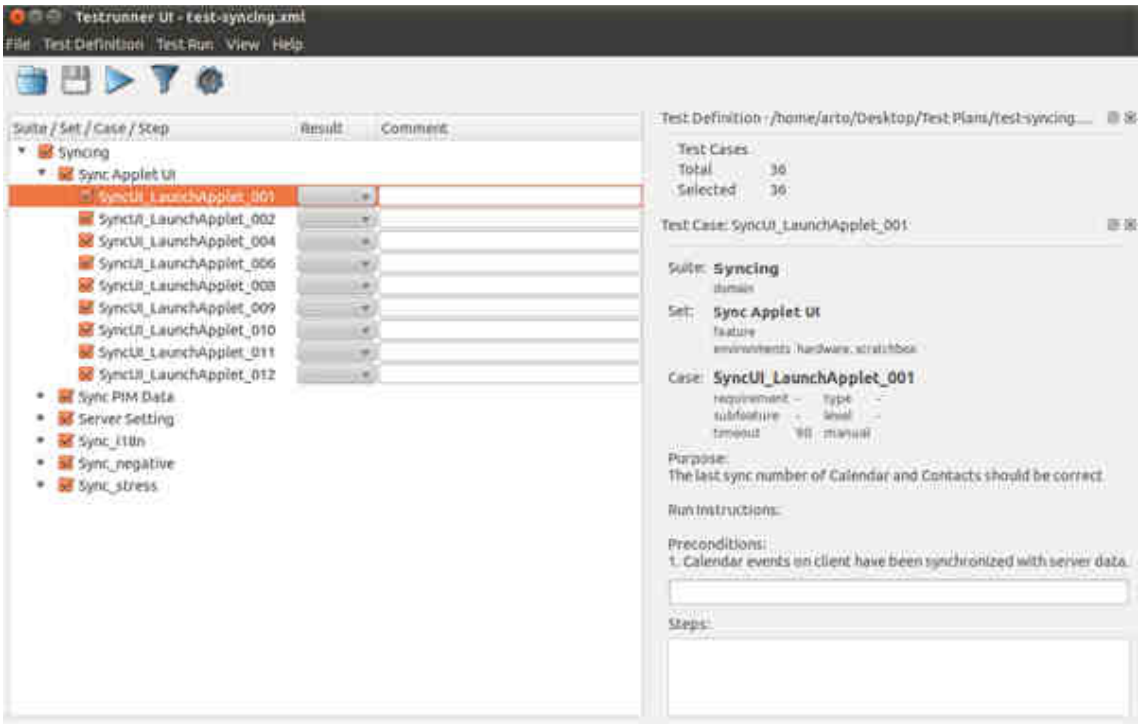


Figure 3.3 Testrunner main view with test plan opened

One advantage of Testrunner is that already run tests can be modified. Command line interface does not provide this feature. Again like whole Testrunner itself this is helpful in manual testing. Test engineer can return to previously run tests and add comments or change result, if there is more to add to the test. After execution Testrunner shows steps' standard and error output in a dock widget. Test engineer also sees other statistics like run time for whole test plan and test case verdicts.

3.4. OTS, Open Test Service

Open Test Service (OTS) is intended for testing software systems on evolving hardware architectures. It was born out of the test framework incrementally developed for use on Maemo project and is used as testing service by Nokia. OTS is fully open-sourced software written in Python using AMQP (Advanced Message Queuing Protocol) and runs on Linux. AMQP is an open standard designed for passing messages between applications [9]. OTS is designed to be a part of continuous integration. High level architecture of OTS is show in Figure 3.4.

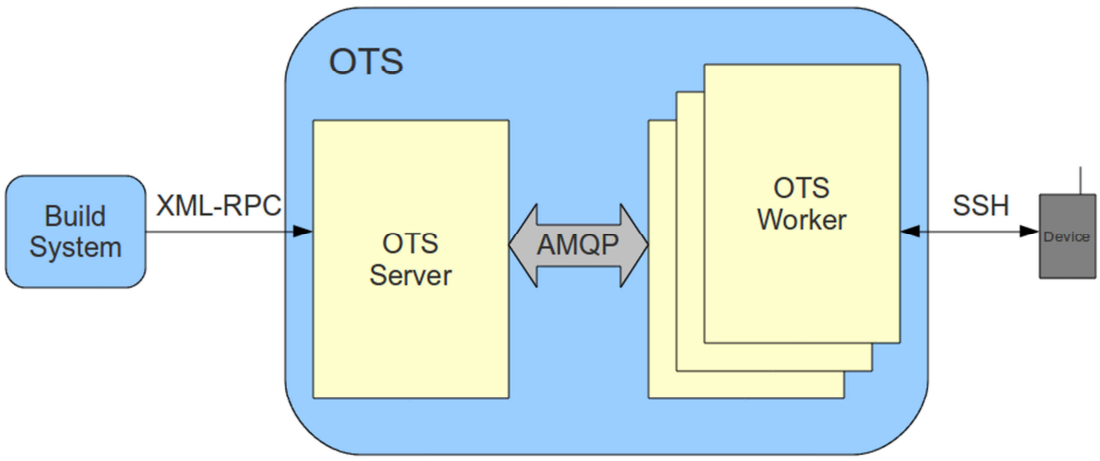


Figure 3.4 OTS architecture

3.4.1. OTS server

Test requests are sent to OTS server using XML-RPC (Extensible Markup Language Remote Procedure Call). XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP (HyperText Transfer Protocol) as transport mechanism [10]. Test requests can also be triggered from command line using *ots_trigger* command line tool but the standard way is that requests are sent by a build system using XML-RPC. The OTS server validates the request and adds it to AMQP messaging queue. These requests contain at least the firmware image that tests will be run on and the list of test packages to be executed. From there suitable OTS worker picks up the request and runs tests and collects results. OTS server is then responsible for publishing the results. Results are published through plugins. The default included publisher plugin sends results to QA Reports and informs via email that request is done. OTS server is usually run on dedicated server machine and can handle large number of requests.

3.4.2. OTS worker

OTS worker is a component that is connected to the device under test. It picks test requests prepared by OTS server from the AMQP queue. The worker formats the device and flashes image provided in the request. The worker has a conductor component that uses a Python component called customflasher that handles this. It also makes sure that device boots properly after flashing and that device is accessible through a SSH connection. The customflasher can be modified to support different devices. After successful flashing operation, the conductor uses Testrunner-lite to execute test packages listed in the request. After test execution, the result XML file and other possible attachments are delivered to OTS server through AMQP messaging queue. OTS worker is capable of handling two test runs on two devices at the same time.

Figure 3.5 shows OTS test flow starting from sending request and ending to result publishing.

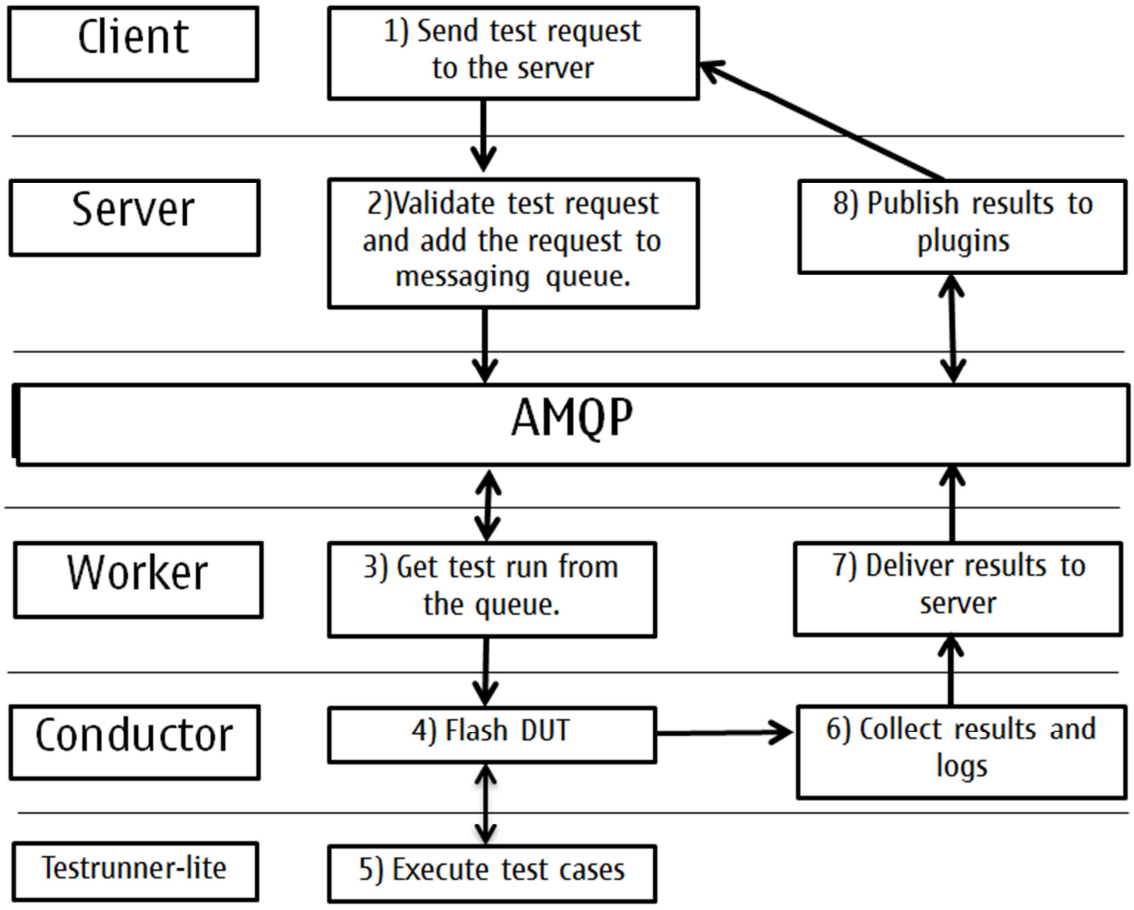


Figure 3.5 OTS test flow

3.5. MeeGo QA Reports

MeeGo QA Reports is open source Ruby-on-Rails test reporting web service. It converts result XML produced by Testrunner-lite to human readable format. Reports can be uploaded manually by testers or automatically by OTS publishing plugin. Manual testers can also add additional info that is not present on the results XML. Info like test objectives, build used, test environment, quality summary and explanations for failed test cases. Results page also shows history information about how test results have changed over time. Results are shown grouped by feature as well as in more detailed view where each test case is shown individually. Detailed report also shows possible measurement results and graph on how the results have changed between test runs. The result page is discussed in more detail later in this thesis.

Test results are hierarchically categorized by release version, target, test type, and HW product. Release version in case of MeeGo could be 1.2 for example. Target refers to the components and deliverables of the MeeGo project. Example of a target and a test type is acceptance testing done on handset. Here the handset is the target and

the acceptance the test type. On the lowest level of the hierarchy, there is the HW product which points to the device under test. Example of such a hierarchy can be seen on Figure 3.6.

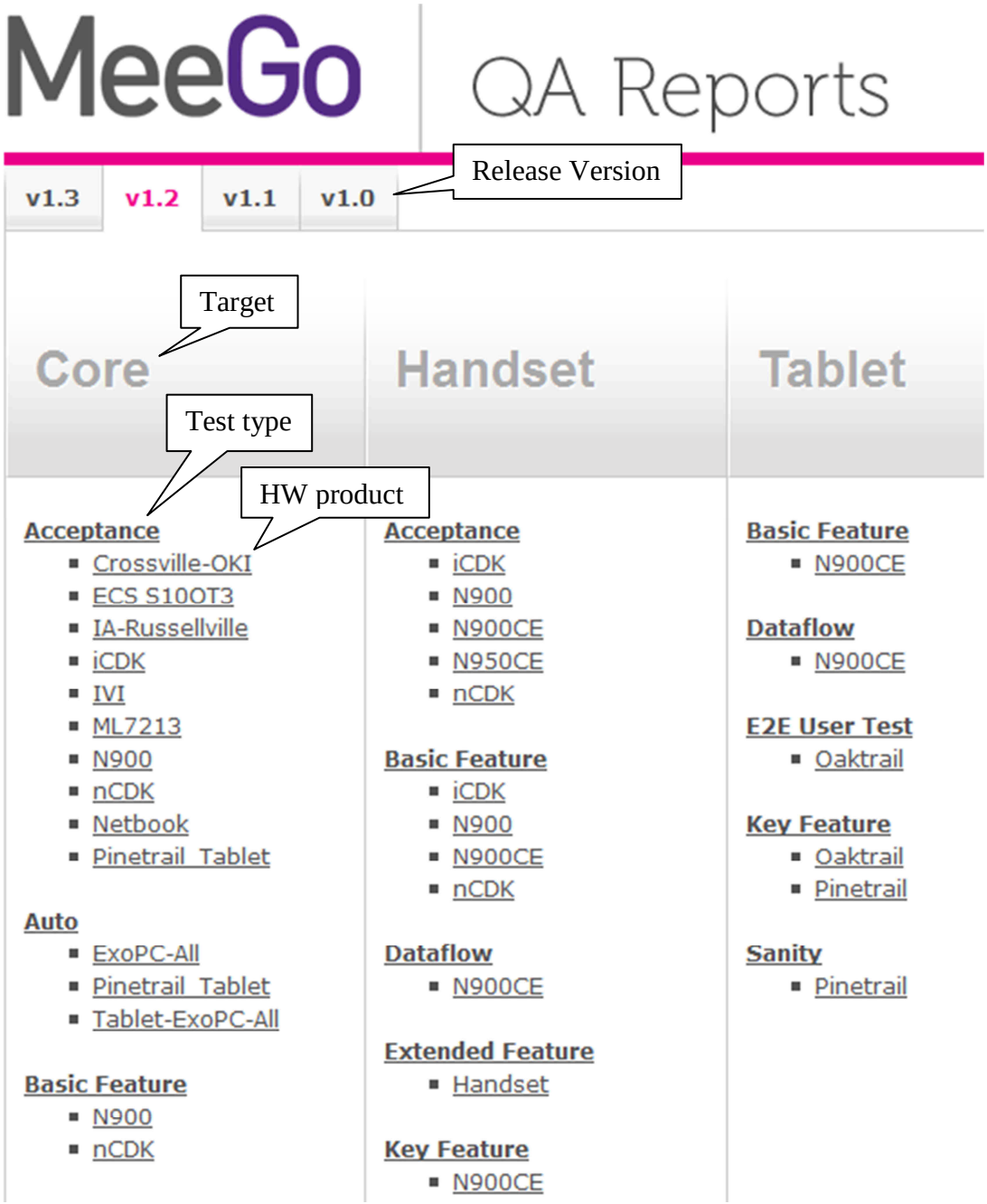


Figure 3.6 Example of test reports categorized hierarchally

3.6. Hardware-environment for test automation system

A standard OTS environment needs a powerful PC for OTS server and, depending on the number of devices under test, varying number of worker PCs. USB switchbox and flashing jig or dummy battery are more specialized hardware needed. These are all the

things needed to run tests. Hardware, like media servers or WLAN access points, can easily be added to support more comprehensive testing.

3.6.1. Worker PC

Worker PC is a regular desktop PC running Linux. It is running OTS worker instance. The device under test is connected to the PC through USB switchbox. Worker PCs do not require much computing power. Ability to run modern Linux operating system (usually some variant of Ubuntu but Fedora is also supported) and two (or three if there is need to connect two devices to the worker) USB ports are the only requirements. Switchbox limits the devices that can be connected to worker PC to two devices. When running test farm of tens of devices it is good to have physically small worker PCs with low heat output and power consumption.

3.6.2. USB switchbox

USB switchbox called OptoFidelity HAT (Hardware Accessory for Testing) is a test hardware that is connected to PC's USB interface. The OptoFidelity HAT board has two USBs and DC power-supply buses, which can be switched on and off independently. The USB data and power lines can be controlled separately. HAT also supports several different sensors with up to 4 sensors supported at the same time [11]. Sensors that can be connected to HAT are acceleration, audio, current, optical, and temperature sensors. There is special open source driver software for Linux called hat-control through which HAT can be controlled. Using this driver, HAT is controlled through USB control cable. For every USB bus, one USB input cable needs to be connected. On the back of switchbox, there is power input for the switchbox itself, two USB inputs and USB control input. On the front there are two USB outputs, two power outputs and 4 sensor inputs. Figure 3.7 shows the switchbox with 2 devices connected.

Figure 3.7 OptoFidelity HAT USB switchbox with two N900 devices running MeeGo connected

3.6.3. Device under test

There are requirements for device that is being tested. It must be possible to switch power on or off with USB switchbox. This means that normal device with battery cannot be used. Usual solution with MeeGo devices is dummy battery or flashing jig. Dummy battery replaces the normal battery of the device and power can be fed from switchbox. Flashing jig also adds support for serial connection to the device which can be useful in debugging. Figure 3.7 has two N900 devices with flashing jig.

4. CREATING TEST CASES AND BUILDING THE ENVIRONMENT FOR TESTING

In this chapter we go through the general structure of the test cases, and tell what test cases where chosen to the set. The chosen test cases are packaged to be usable with OTS environment. Running the performance tests needs extra hardware and software compared to the standard OTS environment. Description of the extra hardware and software added to the standard test automation environment is included. Testrunner-lite's architecture, and adding measurement support for it, is presented next. Finally, we take a look at running the test cases and what kind of results they produce.

4.1. Creating test cases

Test suite used was called MWTS (MiddleWare Test Suite). It provides mostly performance and reliability NFT (Non-Functional Tests) cases. All the test cases used are old test cases that were modified to be usable with test automation. Most test cases were already automatic. What needed to be done was automating the initialization and the teardown of the test cases. This means connecting automatically to i.e. WLAN connection and then after test returning to state before testing. For example, cleaning temporary files and disconnecting from the connection used. Finally, when test cases were automated, a test plan XML file was created and test cases packaged.

4.1.1. Adding test cases to automatic test set

As the test cases were mostly already automatic, converting them to be run fully automated was not very difficult. The actual tests were already ran automatically, but device was initialized manually. The test plan XML file provides pre_steps and post_steps for doing initializing and teardown.

Features and test cases chosen to the set:

- messaging
  - send SMS latency
  - send and receive SMS latency
- WLAN
  - download throughput for open, WEP, WPA TKIP and WPA2AES (5GHz) WLAN connections
  - upload throughput for open, WEP, WPA TKIP and WPA2AES (5GHz) WLAN connections

- CPU benchmarks
  - dhrystone DMIPS
  - whetstone MIPS
  - stream 1 (copy latency, copy bandwidth, scale latency, scale bandwidth, add latency, add bandwidth, triad latency and triad bandwidth)
  - stream 2 (fill latency, fill bandwidth, copy latency, copy bandwidth, daxpy latency, daxpy bandwidth, sum latency and sum bandwitdth)
- USB
  - download throughput
  - upload throughput
- graphics
  - framebuffer full screen buffer write speed
  - OpenGL-Blit with blend and widgets with shadows + particle + rotate + zoom and blur all (framerate, CPU use of test process, CPU use of all processes)
  - OpenGL-Blit with blend and animated widgets with shadows (framerate, CPU use of test process, CPU use of all processes)
  - OpenGL-Blit with blend and widgets with shadow (framerate, CPU use of test process, CPU use of all processes)
  - OpenGL-Vertex shader performance (framerate, CPU use of test process, CPU use of all processes)
- bluetooth
  - RfComm send 10MB throughput
  - L2CAP send 10MB throughput
  - RfComm receive 10MB throughput
  - L2CAP receive 10MB throughput
- video capture
  - camera capture video 480p framerate
  - camera capture video 720p framerate
- still capture
  - camera take photo 7Mpix shot to save latency
  - camera take photo 3Mpix shot to save latency
- core
  - MyDocs FAT write and read speed
  - rootfs ext4 write and read speed
- voice calls
  - Telepathy create call latency
- multimedia
  - local playback video frame rate
  - streaming video frame rate

Small part of the test plan XML file is included in Appendix 1. It shows the test for the voice calls feature.

4.1.2. Basic construction of test cases

MWTS test assets use MIN test framework, which is a test harness for testing Linux non-UI components. MIN can be used for both test case implementation and test case execution [12]. It provides a command line interface and easy-to-use graphical user interface for manual testing. It also provides results, but there are additional log and result files also produced by MWTS test assets themselves. The usage of MIN is made as user friendly as possible. In normal situation, just install MIN and test package and then start testing with MIN. Situations requiring special attention are listed in test asset readme files. Test cases are scripted by MIN scripts and interpreted by the MIN framework script interface. This provides API for using actual test libraries which are written with Qt. Each test asset provides its test MIN scripts, and an user may create new cases simply by writing more cases according to examples. If the user wants to create totally new test asset, or study how MWTS test assets are generally constructed, test asset called mwts-template can be used as a reference.

In case of test-automation MIN scripted test cases are executed using the command line interface of MIN. MIN is invoked from command line using the following command:

min –c –t casename,

where casename is the name of the test case in MIN script files. MIN script files are located in */usr/lib/min* and MIN modules which load these script files are located in */etc/min.d/* folder. Here is an example of one test case inside a script file named *mwts-network-wlan-nft.cfg*:

```
[Test]
title NFT-WLAN-Upload_throughput_WPA2aes-THRO
createx libmin-mwts-network asset
asset Init
asset SetTestTimeout 220000
asset Connect wpa2_aes_ap
loop 7
asset StartIteration
asset ServerStartIperf 180
asset EndIteration
sleep 5000
endloop
asset Disconnect
asset Close
delete asset
[Endtest]
```

Example of a module definition from mwts-network package:

```
[New_Module]
ModuleName=scripter
TestCaseFile=mwts-network-wlan.cfg
TestCaseFile=mwts-network-psd.cfg
TestCaseFile=mwts-network-wlan-nft.cfg
TestCaseFile=mwts-network-psd-nft.cfg
[End_Module]
```

Path where test case files are being searched can be set in */etc/min.d/min.conf*. Default settings:

```
[Engine_Defaults]
ModSearchPath=/usr/lib/min
ModSearchPath=$HOME/.min
[End_Defaults]

[Logger_Defaults]
EmulatorBasePath=/var/log/tests
EmulatorFormat=TXT
EmulatorOutput=TXT
ThreadIdToLogFile=NO
LogLevel=debug
[End_Logger_Defaults]
```

Also the logging level if MIN can be defined here. The MIN log file *min.log* can be found from path defined by *EmulatorBasePath* variable. Usually, logging done by MWTS asset is more important than MIN logging. MWTS logging level can be set by defining *MWTS_DEBUG=1* or *MWTS_LOG_PRINT=1* environment variables. *MWTS_DEBUG* enables debug logging to MWTS log files in */var/log/tests/*. Logs are named *<casename>.log*. *MWTS_LOG_PRINT* variable enables printing logs to standard output which, when used with testrunner-lite, directs log to the result XML file.
In addition to .log files, MWTS asset also produces various other files in */var/log/tests* folder. Result file provides a high level log what happened on the test run. An example of result file from a WLAN upload throughput test:

```
--- Test 'NFT-WLAN-Upload_throughput_WPA2aes-THRO' started
- NetworkTest: Connection latency: 4912 : ms
- NetworkTest: Connect: passed
-- iteration 1 --
- NetworkTest: Upload: 25.166 : Mbits per second
- NetworkTest: Throughput: passed
-- iteration 2 --
- NetworkTest: Upload: 25.0453 : Mbits per second
- NetworkTest: Throughput: passed
-- iteration 3 --
- NetworkTest: Upload: 25.0583 : Mbits per second
- NetworkTest: Throughput: passed
- NetworkTest: Disconnect: passed
```

```
Min: 25.05
Max: 25.17
Mean: 25.09
Stdev: 0.07
Median: 25.06
Target: 0.00
Fail: 0.00
Overall result : PASSED
Memory usage: 409.15+/-0.08 MB
Memory diff: 0.14 MB
```

Other meaningful result file for performance testing is cvsreport file which has the measurement result in a format that Testrunner-lite understands. This format is explained in chapter 4.3. MWTS cases also store information about CPU I/O wait, CPU usage and status of used and free memory during the test run.

4.1.3. Packaging the test cases and creating test plan XML

A test case is packaged as a Debian software package. There are requirements for test case packaging, because of the way OTS runs the tests. Test packages must have name ending in –tests or –benchmark and the test plan XML file must be located in */usr/share/<test package name>/tests.xml*. Reason for this is that OTS is only given a list of test packages to be executed in the test request. As the test plan XML file is not passed with requests, OTS needs to know where to look for it in the device under test. Performance set package can be considered a meta-package which only has a test plan XML file where test cases from various MWTS packages are gathered. MWTS asset packages needed are defined as dependencies which are installed with performance asset's package.

Creating the test plan XML is straight forward after test cases have been picked. The test plan format and commands to run test cases are added inside step elements. Get file elements are used to get produced measurement and log files.

4.2. Building the testing environment

Testing various different WLANs, USB throughput, BT throughput, and video streaming meant that standard OTS worker environment was not enough. Workers needed extra software and also more hardware was added. Four wireless access points and media-server were added to the environment. Because one of the access points had an open WLAN, media-server or any of the access points could not be connected to the Nokia intranet.

Three of the access points had one 2,4GHz WLAN with WEP, WPA or WPA2TKIP encryption. Fourth one had two, the open one and 5GHz WPA2AES. WPA wireless one is not used in performance test set but it was setup for other assets, such as reliability and dataflow, also run on this environment. All the access points were assigned a static

IP and connected to the media-server through standard network switch. The WEP access point had DHCP server on, so all devices connecting to WLAN connections get IP address assigned to them.

Media-server is a standard Ubuntu Linux PC with various extra applications installed. Added software were MIN, Asterix SIP (Session Initiation Protocol) server, Apache HTTP server, FTP (File Transfer Protocol) server, Darwin RTSP (Real Time Stream Protocol) server and Iperf server. HTTP and RTSP server was needed for video streaming cases run as part of the performance set. Iperf server is used in WLAN throughput tests. Iperf was developed to be a modern alternative for measuring maximum TCP and UDP bandwidth performance. FTP server and Asterix SIP server are not needed in performance set but added for other test sets. Asterix SIP server supports a lot of configuration options. For this environment, only two SIP users needed to be created. Other one is used on the device under test side as a user and other one is configured to answer all incoming calls and playing few seconds of music before hanging up.

On the worker side MIN daemon needed to be running to enable Min master-to-slave operations needed for Bluetooth throughput tests. These Bluetooth tests also needed mwts-common, mwts-network and mwts-bluetooth asset packages installed to the worker. Since worker PCs did not have internal Bluetooth chip a Bluetooth dongle was added. Finally worker also needed Iperf server running for USB throughput tests.

4.3. Measurement support

For easy displaying and reporting results through QA Reports, a support for structured measurement series was added to the test plan and Testrunner-lite. The idea is that test asset can produce measurement data in CSV (Comma-Separated Values) format and that can be passed to Testrunner-lite through get file operations in the test plan XML file. Testrunner-lite adds the measurement values to the result XML file, which can then be shown in the reporting tool. Other implemented feature was a support for current measurement using HAT USB switchbox and current sensor. The support was implemented to Testrunner-lite as an experimental feature, and it uses the same format for reporting results as the general measurement support. The support was implemented for the whole tool chain. In this thesis we only deal with the XML format and Testrunner-lite implementation of measurement support. Details were agreed in feature requests in MeeGo bugzilla [13].

4.3.1. Adding measurement support for test plan and result file

As Testrunner-lite already supported get file operations, the easiest way to implement this functionality was adding measurement attribute to the file element. This allows test asset to produce as many measurement results as needed. Example of a measurement result definition:

```
<get>
  <file measurement="true">
    /tmp/blts_gles2_blit+blend+widgets+rotate+zoom.csv
  </file>
</get>
```

When measurement attribute is set as true, the file inside the element is handled as measurement data. Measurement data has the following CSV format:

```
name;value;unit;[target;failure;]
```

Name and unit are strings, and value, target, and failure floating point numbers. Target and failure are optional. If they are specified, the measurement can affect the test case result. An example of CSV file with measurement data:

```
framerate;6.992156;1/s;
cpu_use_test_process;10.617490;%;
cpu_use_all_processes;11.788212;%;
```

Results of multiple measurements can be defined to the CSV file, unless measurement results are in series format. When measurement consists of series of data, series attribute needs to be used in the file element:

```
<file measurement="true"
series="true">/path/to/measurement/series.txt</file>
```

Series data only consist of values of single measurement

```
name;unit[;target;failure]
[yyyy-mm-ddThh:mm:ss[.ssssss];]value
[yyyy-mm-ddThh:mm:ss[.ssssss];]value
[yyyy-mm-ddThh:mm:ss[.ssssss];]value
...
```

Similar to the measurement results only providing one value, name and unit are strings, and target and failure floating point numbers. The difference is that there are multiple values that can be time stamped. This allows graphs to be drawn from measurement results. Single and series measurements need to be shown differently in the result file. Single measurement examples from OpenGL tests:

```
<measurement name="framerate" value="7.055881" unit="1/s"/>
<measurement name="cpu_use_test_process" value="10.488880"
unit="%"/>
<measurement name="cpu_use_all_processes" value="12.277228"
unit="%"/>
```

Series measurement example:

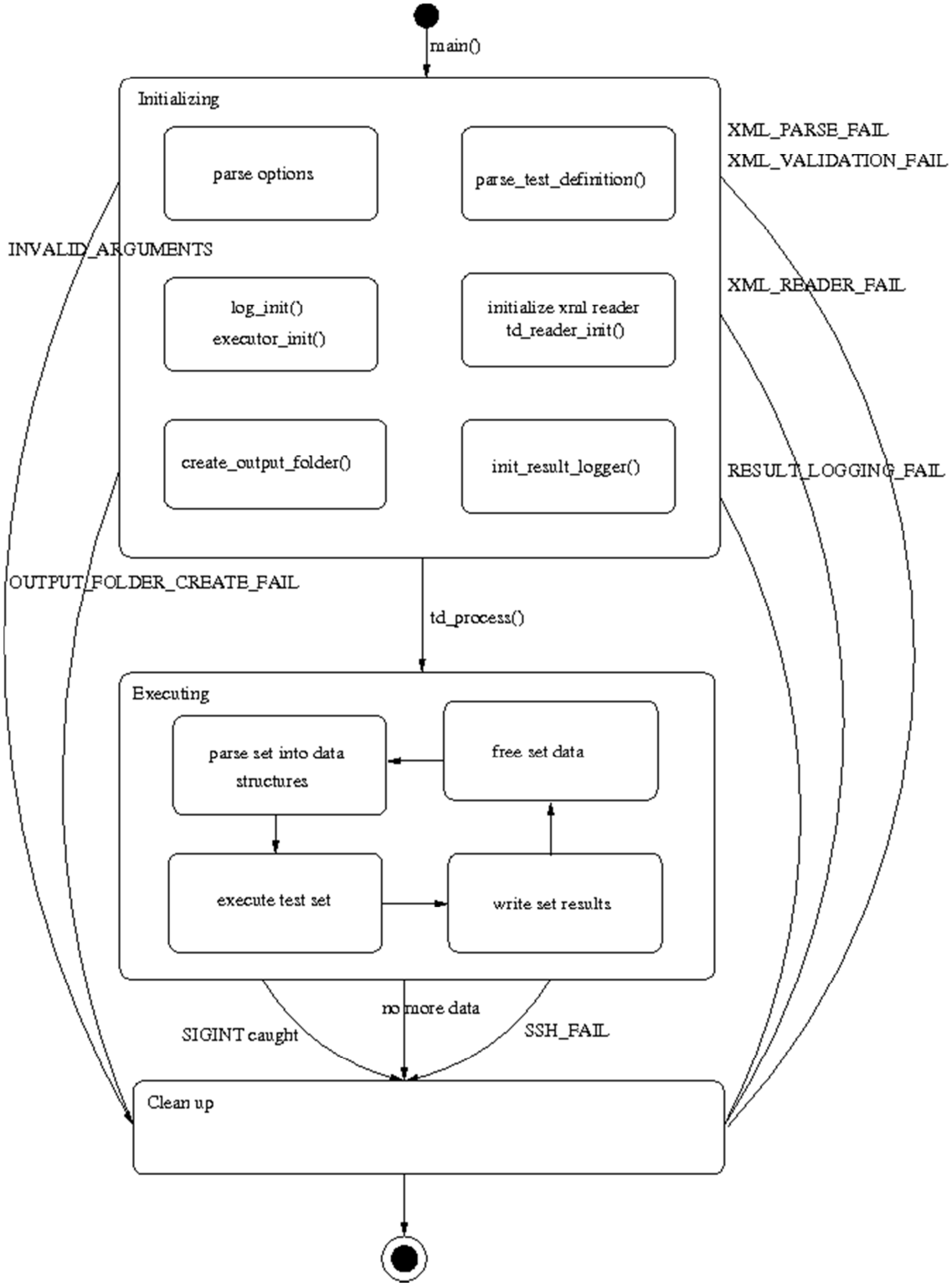
```
<measurement name="Average current" value="22.4" unit="mA"/>
<series name="Current consumption" unit="mA">
  <measurement timestamp="2011-02-24T14:28:02.123456"
value="22.8"/>
  <measurement timestamp="2011-02-24T14:28:02.223456"
value="22.0"/>
</series>
```

As a special case there is current measurement support done by Testrunner-lite which uses interval_unit and interval attributes in the measurement element. Example from current measurement test output:

```
<series name="Current samples" group="Current measurement"
unit="mA" interval="200" interval_unit="ms">
  <measurement value="183.000000"/>
  <measurement value="117.700000"/>
  <measurement value="132.300000"/>
  <measurement value="158.600000"/>
  …
  <measurement value="6.100000"/>
</series>
```

4.3.2. Architecture of testrunner-lite

Testrunner-lite is a C program consisting of roughly 13400 lines of code (including comments and empty lines). On the top of this, there are unit and regression tests. The program flow consists of three phases: initializing, executing, and clean up. Picture 4.1 shows the simplified program flow of Testrunner-lite.



Picture 4.1 state machine of program flow of Testrunner-lite

Main modules of Testrunner-lite are main, executor, test definition parser, test definition processor, test definition data types, logger, and test results writer. The initializing part is handled by the main module. It starts by parsing command line arguments and initializes logger and executor modules. After this, the test plan XML file is read by the test definition parser module. The main module exits if, the test plan XML file has

syntax errors or does not pass validation against schema. After XML is validated the main module initializes XML reader and test results writer, before test definition processor module continues the execution.

The test definition processor communicates with the parser module. It calls function called *td_next_node* as long as there is data to be parsed in the test plan XML file. The test plan is parsed one set at a time into data structures defined in test definition data types module. Decision to only parse one set at a time was made to conserve memory, if Testrunner-lite is run on low-power device. After parser module has a set parsed into internal data structures, it uses callbacks in the processor module to pass the set to the processor. For actual test execution, the processor module uses executor module. The executor module takes care of test step execution. The module is divided to 2 parts, the execution is directed to either remote executor or manual executor based on the test plan XML file. Finally, when tests are executed successfully, the processor uses test result writer module to write results of the whole set to result XML file. If there are more sets to be read, the old set is freed and new set is read to data structures. When all sets have been processed, all the allocated memory is freed and Testrunner-lite exits.

The logger module is responsible for logging everything going on during test run. Logging level can be set to info or debug. The info level logging shows standard info messages as well as possible errors and warning. The debug level logging adds a lot of debug messages to the output and is not generally used with OTS. Logging can be done to stdout or through HTTP logger which is used with OTS. This allows OTS server to show log messages of Testrunner-lite also in the test run log page.

In addition to these, there are event, hardware info, test filters, and utilities modules. The event module provides experimental event support to Testrunner-lite. It allows two Testrunner-lites to communicate with each other through AMQP messaging queue. This allows, for example, testing phone call between two devices. Testrunner-lite number one knows to expect a specific event from Testrunner-lite number two after the phone call is made. After it receives the event, it can check, if there is phone call incoming, send response to Testrunner-lite number one, and evaluate the test case result accordingly. Hardware info module is a device specific part which is used to query info about the device under test before tests are run. Currently, the test plan format supports hwproduct and hwbuild attributes to be defined in the start element of result XML file. The test filters module has various functions to help parse Testrunner-lite filter options that can be defined when calling Testrunner-lite. The utilities module contains helper functions, for example, string parsing and validating that output from test process is valid UTF-8 characters.

4.3.3. Modifying Testrunner-lite

Changes were needed to the test definition parser, the test definition processor, the test definition data types, and the test result writer modules. Also a new module called test measurement was added. XML reading is done using libxml2 library.

The test definition parser needed to be able to read new attributes, measurement and series, in get file element. Function that handles get file elements is called *td_parse_gets*. It only needed few more lines to read the new attributes and add them to libxml2 librarys *xmlList* structure. New data types for measurement, measurement series, and measurement item in series were added to test definition data types.

```
/** Test measurement */
typedef struct {
 xmlChar *name;              /**< E.g. bt.upload */
 xmlChar *group;             /**< E.g. bt_measurements */
 double   value;             /**< Value of measurement */
 xmlChar *unit;              /**< E.g. Mb/s */
 int      target_specified;  /**< Is target and failure
specified ? */
 double   target;            /**< Target value */
 double   failure;           /**< Failure value */
} td_measurement;
/* ------------------------------------------------ */
/** Test measurement item */
typedef struct {
 double   value;             /**< Value of measurement */
 int      has_timestamp;     /**< 1 if timestamp set,
otherwise 0 */
 struct timespec timestamp;  /**< Timestamp of the
measurement */
} td_measurement_item;
/* ------------------------------------------------ */
/** Test measurement series */
typedef struct {
 xmlChar *name;              /**< Name of series */
 xmlChar *group;             /**< Measurement group */
 xmlChar *unit;              /**< Unit of series */
 xmlListPtr items;           /**< measurement item series
*/
 int      target_specified;  /**< Is target and failure
specified ? */
 double   target;            /**< Target value */
 double   failure;           /**< Failure value */
 int      has_interval;      /**< 1 if interval set,
otherwise 0 */
 int      interval;          /**< Interval value */
 xmlChar *interval_unit;     /**< Interval unit */
} td_measurement_series;
```

Other than just definitions of data types, this module also allocates and deletes these structures. Functions to allocate and free measurements, measurement series, and measurement items were added.

Before measurement feature was added Testrunner-lite only supported get file elements in the set level. As measurements were intended to be on case level, support for get file to case was added. Set level function that processes get file elements was called *process_get*. New function called *process_get_case* was added. It first uses

process_get to obtain the files from device under test and then, if get file was marked as measurement, uses the new test measurement module to parse measurement data from the file. Test measurement module has the following public API:

```
/* FUNCTION PROTOTYPES */
/* ------------------------------------------------ */
int get_measurements (const char *file, td_case *c, int
series);
/* ------------------------------------------------ */
int eval_measurements (td_case *c, int *verdict,
             char **fail_string, int series);
/* ------------------------------------------------ */
int process_current_measurement(const char *filename,
td_case *c);
/* ------------------------------------------------ */
```

The *get_measurement* function reads the measurement file and adds measurements or measurements series to data structures. Function *eval_measurements* compares measurement results to target and failure values and gives verdict based on that. Function *process_current_measurement* is used in the experimental current measurement support, more on that later.

Finally, support for writing the results to result XML was added. This meant changes to the test results writer. As other parts of Testrunner-lite process results into easy to use data structures, implementing results writing was straight forward.

```
LOCAL int xml_write_measurement (const void *, const void
*);
/* ------------------------------------------------ */
LOCAL int xml_write_series (const void *, const void *);
/* ------------------------------------------------ */
LOCAL int xml_write_measurement_item (const void *, const
void *);
```

Each function writes the part its names after to results file. Function *xml_write_measurement* writes single measurement results, *xml_write_series* writes start element of measurement series and then uses *xml_write_measurement_item* to write measurement values to the series.

The experimental current measurement support was implemented using HAT USB switchbox and a special sensor for current measuring. Current measurement is started at the start of a case and it uses HAT driver software to produce measurement file. HAT driver software consists of two parts: *hat_drv* and *hat_ctrl* programs. *Hat_drv* is the driver that control HAT box through USB and *hat_ctrl* is the control software that controls the driver. Communication between the driver and control processes is done using shared memory. Control software allows measurement data from current sensor to be written to file. The sensor input port where sensor is installed, sample rate and type

of sensor outputting the data can be chosen. Testrunner-lite parses the output file and adds measurement data to the result XML.

Adding new features to Testrunner-lite, like support for measurements, is rather easy thanks to the well thought architecture. Including comments and empty lines the measurement feature added about 700 lines of code. The use of C as implementation language makes string operations and communication with OTS difficult. Porting Testrunner-lite back to Python should be studied.

4.4. Running tests and reporting the results

The test cases are run using OTS and the results will be reported to QA Reports. Tests are scheduled to run weekly. Test reports are analyzed and tests run manually, if there are unclear failures.

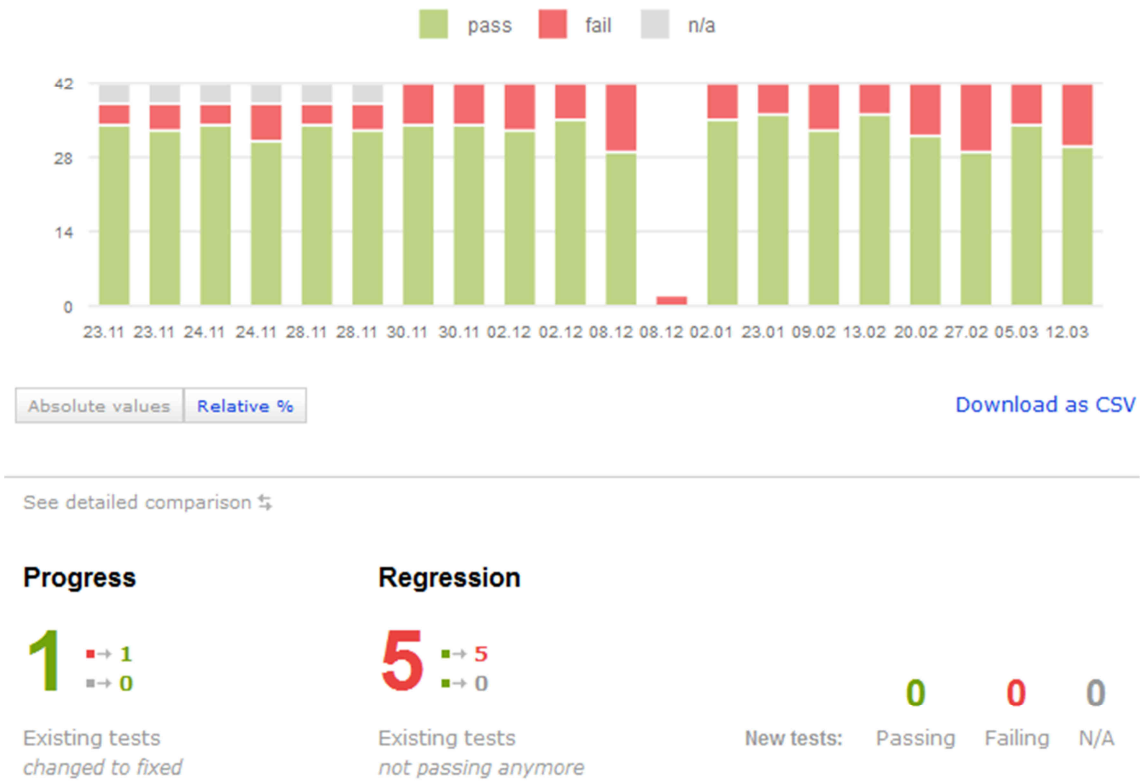
4.4.1. Executing tests

Tests need to be released to repositories so they can be run with OTS. An integration request is similar to a test run request, and is done through build system. The test package is sent to the staging queue and from there accepted to the actual repositories. The build system can pick the test packages from repositories when building the image. When image is ready the standard OTS execution flow applies.

Tests are scheduled to run using dashboards. Dashboards are build systems way to implement VCS hooks, that are useful when integrating new packages, or to schedule test runs using crontab syntax. For performance measurements the dashboard is called System Non-Functional Testing (Performance) and it only runs scheduled tests. Integration is done using another dashboard that handles integration of all the new MWTS asset packages. All that the dashboard needs is the definition of the test request and the schedule set by crontab syntax. Dashboards are part of the not public proprietary system. MeeGo has similar concept called BOSS (Build Orchestration Supervision System) but it never really materialized before the death of open MeeGo.

4.4.2. Reporting the results

After the test run completes owner and anyone on the cc list of a dashboard is notified through email about the results. Notification includes links to the test request and QA Reports result page. Pass or fail verdict is included in the email. Results can also be browsed from QA Reports front page. After selecting the right test type and the hardware product from QA Reports, it shows history data about the tests. This view is shown in Picture 4.2

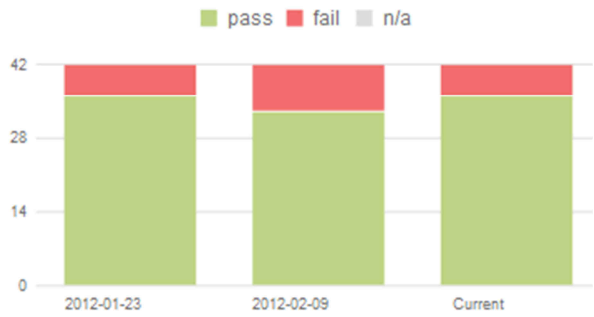


Picture 4.2 Progress and regression between test runs

Below the history data there are the individual test runs. They show more detailed results about the test run. Page starts with general information about the test run like test objective, image used, test environment and quality summary. Below this there are results summary shown in Picture 4.3.

Result Summary

Total test cases	42	
Passed	36	↑ +3
Failed	6	↓ -3
N/A	0	
Run rate	100%	
Pass rate of total	86%	↑ +7%
Pass rate of executed	86%	↑ +7%
NFT Index	0%	

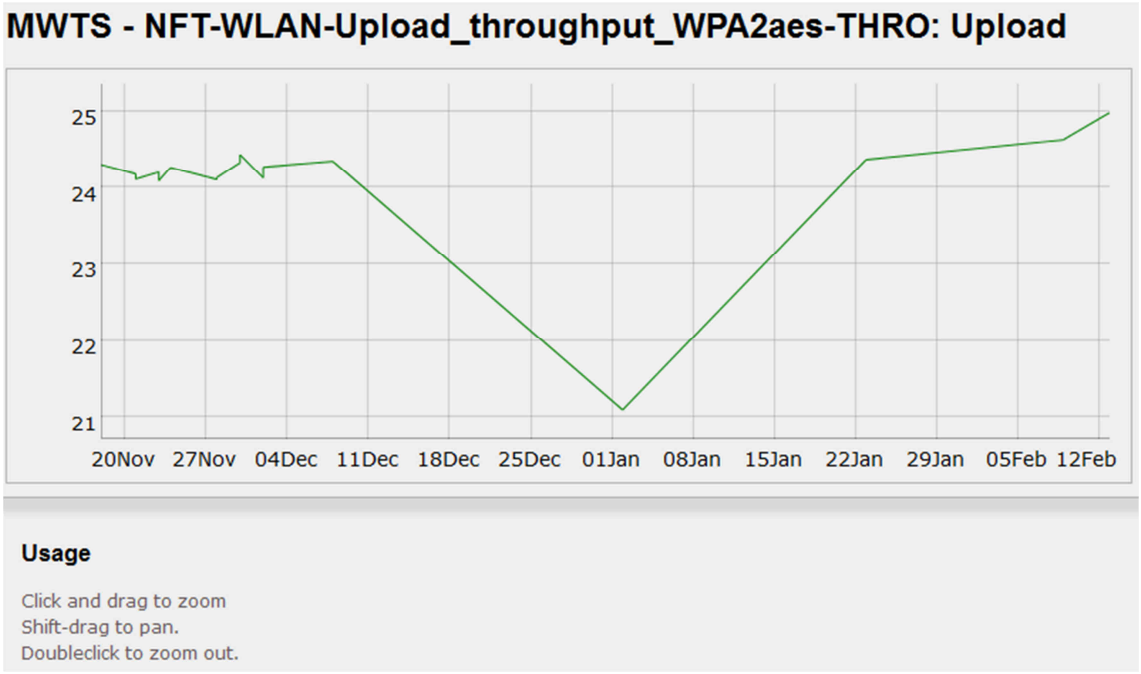


Test Results by Feature

Feature	Total	Passed	Failed	N/A	Pass%
Messaging	2	2	0	0	100%
WLAN	8	8	0	0	100%
CPU Benchmarks	4	4	0	0	100%
USB	2	2	0	0	100%
Graphics	5	5	0	0	100%
Bluetooth	4	0	4	0	0%
Video Capture	2	0	2	0	0%
Still Capture	2	2	0	0	100%
Core	2	2	0	0	100%
Voice Calls	1	1	0	0	100%
Multimedia	10	10	0	0	100%

Picture 4.3 Results summary and test results grouped by feature

After this general summary there are the individual test case results of each test set. In addition to test case verdicts measurement results are shown. Measurement name, value, target, fail limit, and how many percent of the target value was achieved are shown. Next to measurement value there is button to see graph about how the value has changed between test runs. Example of WLAN throughput measurement history graph is shown in Picture 4.4.



Picture 4.4 How upload throughput has changed between test runs.

User is able to zoom and pan the view. Putting mouse over the graph shows exact results of test run as a tooltip. Finally there are all the attachments downloaded using get file elements and other attachments that OTS produces.

4.4.3. Fulfilling the goals

As the MeeGo project is coming to a close the goal was to automate as much as possible. This performance and other assets are meant to provide support to the few test engineers that are left to do MeeGo maintenance and quality assurance. Environment and test cases are up and running so the project can be considered a success.

Another goal was to make people more familiar with the MeeGo QA tools by introducing what is needed to run fully automated tests. These test cases also provide good reference to the future projects. As all the tools shown here are open source, it is possible for open source community to pick up either these test cases or tools.

4.4.4. Analyzing the results

Running the automatic performance test set, including image creation and device flashing takes about two hours. The actual testing part takes an hour and 40 minutes. Reliability level of build system and OTS is on a good level. OTS has been used in hundreds of thousands test runs over the course of several years, and the probability that device gets flashed properly and tests are executed without OTS related problems is over 95% [14]. The actual test case results are not that reliable. Often problems are caused by timing issues, unfamiliarity with automated testing and capacity issues in network connections. In case of the performance test set results are stable and only problems are with the Bluetooth tests. These problems are caused by faulty Bluetooth

dongles that have problems setting power mode on. Packet data performance test was also removed from the set because running performance tests against live network does not produce reliable results. CMU universal radio tester would produce accurate results but due to only couple of them being available and high cost it was not used.

Adding measurement support for the QA tools tool chain works reliably. Testrunner-lite, like OTS, has been stable for years and adding new features did not change this. Format used to show the measurement data is flexible enough and provides all the fields needed.

5. CONCLUSIONS

In this thesis performance test asset, and measurement support to Testrunner-lite and test plan format were created. First, the format of measurement support in test plan was agreed, and then support was implemented to Testrunner-lite and other tools. Then test cases were chosen to performance test asset and the cases modified to be fully automated, and to use the measurement support format of test plan. Building the test environment to run the automated tests was also part of this thesis.

Work was started by adding the support for measurement elements in the test plan and the result file format. This made it possible to show different kinds of measurement results. Testrunner-lite was modified to gather the results from test assets and output the data in the result file. Reporting tool shows the gathered measurement data in human readable format and can be used to analyze the results. The whole tool chain now supports measurement data and can be used by everyone involved in the MeeGo project.

Other part of this thesis was creating the test set and the environment to run the test set. Test environment was done by adding hardware and software to the existing OTS test automation environment. Test asset developed is meant to show, if changes happen in the measured performance values of the system. Already existing test cases were converted to be fully automatic that can be run without the need of human interaction. Teams responsible for maintenance and quality assurance of the system can see, if there are changes in the results and investigate if necessary. Performance test asset is run weekly for weekly release. The test environment created is also used to run other testing that needs the extra hardware and software provided. For example, video streaming cases need server to stream from.

Work done on this thesis can be used in the future projects either inside Nokia or by the open source community. MeeGo QA tools are freely available to everybody. Although the performance asset described here is not available as open source, the test cases used to piece together this test set are open sourced.

REFERENCES

[1] Haikala, I. & Märijärvi, J. 2002. Ohjelmistotuotanto. 9th printing. Pieksämäki, RT-Print.

[2] Binder, R. 2001. Testing object-oriented systems: models, patterns, and tools. 3rd printing. Addison-Wesley.

[3] Dustin, E., Rashka, J. & Paul, J. 2001. Automated Software Testing. 5th printing. Addison-Wesley.

[4] Fowler, M. 2006. [WWW] [Cited 18.3.2012]. Available at http://www.martinfowler.com/articles/continuousIntegration.html .

[5] Meier, J.D., Farre, C., Bansode, P., Barber, S., Rea, D. 2007. Performance Testing Guidance for Web Applications [WWW] [Cited 18.3.2012]. Available at http://perftestingguide.codeplex.com/ .

[6] MeeGo architecture domain view [WWW] [Cited 18.3.2012]. Available at https://meego.com/developers/meego-architecture/meego-architecture-domain-view .

[7] MeeGo QA verdict definition [WWW] [Cited 18.3.2012]. Available at http://wiki.meego.com/Quality/Glossary#Test_case_verdict .

[8] Chroot and Host Testing [WWW] [Cited 18.3.2012]. Nokia internal document.

[9] AMQP about page [WWW] [Cited 18.3.2012]. Available at http://www.amqp.org/about/what .

[10] XML-RPC homepage [WWW] [Cited 18.3.2012]. Available at http://xmlrpc.scripting.com/default.html .

[11] OptoFidelity HAT datasheet [WWW] [Cited 18.3.2012]. Available at http://www.optofidelity.com/media/Datalehdet/OF_Datasheet_HAT.pdf .

[12] MIN Test Framework User's Guide v1.3 [WWW] [Cited 18.3.2012]. Available at http://min.sourceforge.net/doc/min.html .

[13] Measurement value support in testing tools [WWW] [Cited 18.3.2012] Available at https://bugs.meego.com/show_bug.cgi?id=12980 .

[14] OTS monitoring data [WWW] [Cited 18.3.2012] Nokia internal document.

APPENDIX 1: PART OF THE TEST PLAN XML USED IN PERFORMANCE TESTING

```
<?xml version='1.0' encoding='UTF-8'?>
<testdefinition version="1.0">
  <suite timeout="180" name="mwts-perf" domain="AutomatedTesting">
    <set feature="Voice Calls" name="Voice Calls">
      <case timeout="240" name="MWTS - NFT-Telepathy-Create_call-
LATE">
        <step>source /tmp/session_bus_address.user; MWTS_LOG_PRINT=1
MWTS_DEBUG=1 min -x /usr/lib/min/scripter.so:/usr/lib/min/mwts-
perf.min -c -t NFT-Telepathy-Create_call-LATE</step>
        <get>
          <file measurement="true">/var/log/tests/NFT-Telepathy-
Create_call-LATE.csvreport</file>
          <file measurement="false">/var/log/tests/NFT-Telepathy-
Create_call-LATE.log</file>
          <file measurement="false">/var/log/tests/NFT-Telepathy-
Create_call-LATE.result</file>
        </get>
      </case>

      <pre_steps>
        <!-- Workaround for min hangs with many loaded modules -->
        <step>mkdir -p /tmp/min_confs ; mv /etc/min.d/mwts-*.conf
/tmp/min_confs/</step>
      </pre_steps>
      <post_steps>
        <step>mv /tmp/min_confs/* /etc/min.d/ ; rmdir
/tmp/min_confs</step>
      </post_steps>

    </set>
  </suite>
</testdefinition>
```